A First Set of Design Patterns for Algorithm Animation

Guido Rößling
CS Department, TU Darmstadt
Hochschulstr. 10
64289 Darmstadt, Germany

roessling@acm.org

Abstract

Design Patterns are extremely helpful in preventing programmers from "reinventing the wheel". However, the algorithm animation area does not yet seem to have any Design Patterns, although there are several design issues that have to be resolved in many systems. We present two Design Patterns that address two central points in flexible algorithm animation systems: reverse playing and conceptual uncoupling to allow for easy extension.

1 Introduction

Design Patterns in the Computer Science context go back to the book by Gamma et al. (1995) which mapped the original concept of the architect Christopher Alexander to software development. The book provides approaches for solving problems that may come up in several different applications by following the same basic "idea" or approach, typically using a set of cooperating classes.

For algorithm visualization (in the following, abbreviated as AV), we could not find a "real" description of design patterns in use, although the problems faced by many systems are also similar. For example, users will often find it interesting or even important to be able to step backwards through the visualization (Anderson and Naps, 2001), without (for them) arbitrary limitations, such as a limited undo stack. Additionally, the process of stepping backwards should be reasonably fast.

Since no AV system is likely ever to be really "complete", means for introducing extensions or reconfigurations need to be provided. An interested developer should find it relatively easy to include a new primitive or data structure, provide a new animation effect or state transition, or do both. However, this should be possible without forcing the developer to take care of everything at the same time. Additionally, the developer may not have—or even want to have—a deeper understanding of the underlying system. Therefore, he or she should not be forced to modify existing components, as this decreases the motivation to "provide just a small change", and also increases the risk of making parts of the systems unusable.

In this paper, we define two initial AV patterns, in the hope that they will serve as a first "stepping stone" for the definition and refinement of additional AV-related design patterns.

Section 2 briefly summarizes the main elements of design patterns. Section 3 presents a pattern for easy navigation in both directions without the need of an undo stack. Section 4 introduces a pattern for enabling the extension of different aspects of an AV system without touching other parts. Section 5 summarizes the patterns in this paper and outlines future AV-related pattern research aspects.

2 An Extremely Brief Description of Design Patterns

Design patterns describe problems that occur many times in different parts of our environment. By describing the core of the solution to the problem, the same basic approach can be used to solve the problem, although the actual code is most likely different each time (Gamma et al., 1995, p. 2). A Design Pattern, as described in the basic book by Gamma et al. (1995), has five essential elements:

The pattern name is used to refer to the pattern. It provides a common understanding of what is referred to, assuming that all readers are familiar with the given pattern.

The intent describes the intention of the pattern in a single sentence.

The problem is a description of the situation that is addressed by the pattern.

The solution describes the components that can be used to solve the problem. It does not describe a concrete implementation, but rather the elements that are used to reach a concrete implementation, to allow for easier reuse and adaptation.

The consequences describe the results and trade-offs of applying the pattern. While the use of a pattern may increase the flexibility of the software, it may also affect the runtime or the amount of memory needed.

The book by Gamma et al. (1995) lists many other aspects of a pattern that can be listed, such as the *motivation* and *sample code*. However, we will not strictly adhere to the format for the sake of clarity and brevity.

We will use the pattern name as the name for the sections. The other elements will appear as subsections.

3 Reverse by Fast Forward

3.1 Intent

Reverse by Fast Forward supports flexible unbounded bidirectional navigation.

3.2 Problem

During a visualization of an algorithm, a user may become confused at some stage. Additionally, the visualization of a user-written algorithm may show a bug that needs to be tracked backed to its origin. Both situations benefit from the ability to easily step back to the previous step or steps. Here, a *step* is taken to be a closed set of operations that happen at the same basic time and represent complete actions. Thus, one move of an object along a line will always be part of exactly one step, even if it is rendered using a set of intermediate animation frames. However, the next step may contain another move of the same object.

The standard approach of supporting stepping backwards is to keep an undo stack of previous visualization states. In some systems, this may be a stack of static images, while other systems need to store a complete object representation. Both situations are usually limited by the amount of memory reserved for the stack, and may be further reduced by the size of the shown content. For example, an animation that covers 200 animation steps may require the storing of 199 previous step states for undo. If intermediate animation frames are to be stored, the number rises very quickly. The same is true if the undo stack tries to mirror the user's movement through the animation, and thus may have to store the same intermediate state multiple times as the user steps back through the animation.

The importance of being able to navigate back to a well-understood step in the display is also discussed in research papers (Rößling and Naps, 2002; Rößling and Naps, 2002). It is, however, not trivial to find a good solution for fast and arbitrary navigation, as shown by the statement that "efficient rewind [is] one of the most 'open questions' in AV" (Anderson and Naps, 2001).

3.3 Solution

We use a technique called "reverse by fast forward", a term coined in a discussion between the author and Amruth Kumar during a SIGCSE session break some years ago.

The main limitations of the classical undo stack have been described above: it may reach the fixed memory limit quickly, consumes much memory, and may redundantly store the same set of objects in different positions. Our proposal may seem counter-intuitive at first:

we navigate backwards by quickly moving forwards from a well-defined position in the AV contents. A similar approach is also used in reverse debugging and checkpointing (Boothe, 2000).

For this approach to work, the following conditions must be met:

- The content must have a certain structure, such as separate steps, to allow for a meaningful definition of "current" and "previous step", as well as for the "start" of the contents.
- The objects and transformations must be encoded in a way that allows executing them multiple times, always producing the same results. In practice, it is enough if a copy of the operations is stored even after they have been executed. "Execute and forget"-like operations, such as in the AV system *JAWAA* (Akingbade et al., 2003), which parse the current command, execute it, and then forget about it, are not suited for this approach.
- Two sets of objects must be stored: the original objects as they were initially defined in the animation, and one set of clones of these objects. Thus, the approach takes twice as much memory for storing objects as the normal approach would require.
- The transformation of the graphical objects can be visualized by the system, but can also be executed "quickly" without executing any visualization code.

Instead of executing a given operation or animation step on the original objects, the system will perform the following steps:

1. Determine the animation step the user wants to reach.
2. Ensure that the chosen step actually exists. In the case of manual step input, the user might provide a step number that does not exist, or may navigate forwards beyond the end or backwards beyond the start of the animation. If the chosen step does not exist, stop at this stage.
3. Clone all original graphical objects and place them in an appropriate data structure, such as a hashtable or list.
4. For all steps between the initial state and the target step, quickly perform all transformations on the cloned graphical objects without visualizing the effects.
5. Once the target step is reached, resume normal operation.

In most cases, executing the actual transformations on the objects without creating any visualization contents or updating the display will be much faster than the visualization. Practical experience with the Animal system (Rößling and Freisleben, 2002) shows that the gap between pressing the "go backward" or "go forward" button and the update of the display is usually not or only barely noticeable, even for large animations.

To increase the performance of the display, the clones from the previous animation step may also be stored. If the user requests the next step to be displayed, the first four steps of the item list can be skipped and execution can directly continue. Additionally, snapshots of steps at certain intervals (e.g., every ten steps) can also be taken to support faster navigation. However, this will also increase the amount of memory needed, and may severely harm the animation speed if the user is actively editing the animation, forcing the system to continuously update the "snapshots" - a situation that does not occur in the other application areas (Boothe, 2000).

Arbitrary animation steps can also be dynamically performed in reverse direction if the animation effects are coded appropriately. This is used in the Animal AV system (Rößling

and Freisleben, 2002) to allow fully flexible bidirectional navigation even inside steps, letting "objects fly backwards".

As the pattern requires only the storage of the original and the cloned objects, no UML diagram is given.

3.4 Consequences

The *Reverse by Fast Forward* pattern has a set of consequences:

- The users can always jump to any arbitrary point in the animation. They are not restricted to the *next* or *previous* step, predefined points (such as *start, end*) or a limited number of steps to be reversed.
- Even object-destroying operations, such as scaling by a factor of 0 or multiplying by 0 can be "reversed", as the previous state of the object is retained through the cloning approach.
- The amount of memory used by the application increases, as each graphical object has to be stored twice. As the clones have to be prepared at each step (see 3. in the list above), this may also lead to increased memory fragmentation and garbage collection, which can impact the runtime.

4 Request Handler

4.1 Intent

Request Handler decouples animation effects from graphical objects, making both more flexible and easier to extend.

4.2 Problem

In AV and regular graphics systems, the user typically has (at least) two different abstractions: the *graphical objects* and the *animation* or *editing effects*. Graphical objects may be primitive, such as a text or square, or complex, such as an array or a tree. They may also encapsulate special semantics for some operations, or provide object-specific behavior. For example, changing the fill color of a circle representing a traffic light to red is different from just coloring a "normal" circle; swapping elements requires an appropriate visual rendition to be easily visible to the end-user. Typically, the graphical objects are responsible for storing their current state and can be requested to paint themselves. Animation effects are responsible for changing the graphical objects, optionally also using timing specifications such as the time to wait before the effect starts or its total duration.

Interested developers should be able to implement a new graphical object without touching the existing animation effect. They should also be able to add a new animation effect without having to modify the existing graphical objects, and should be able to avoid providing a hard link between these entities. Finally, existing animation effects that differ only in small aspects should be modifiable without having to touch the animation effect. For example, if an animation effect for changing a color exists, there should be no need to implement a "fill color" change effect, or even modify the existing animation effect.

A standard approach is to incorporate a design such as *MVC* (Model, View, Controller), where the *Model* is the graphical object, the *Controller* role is assumed by the animation effect, and the *View* is the graphical rendition of the object. However, this does not provide the necessary uncoupling described above.

The intention of the described pattern is also similar to the "expression problem" described by Odersky and Zenger (2005). However, their approach (and other related approaches) requires special mixin features that are not available in Java. Related other approaches as

in (Torgersen, 2004) require Java generics, and thus an explicit implementation in separate classes.

4.3 Solution

Use a *Request Handler* class to intercept the direct interactions between graphical objects and transformation effects, as shown in Figure 1. Here, the *ActualModel* entity represents the graphical object. It is aware of its current state and can therefore answer requests by the associated *RequestHandler*.

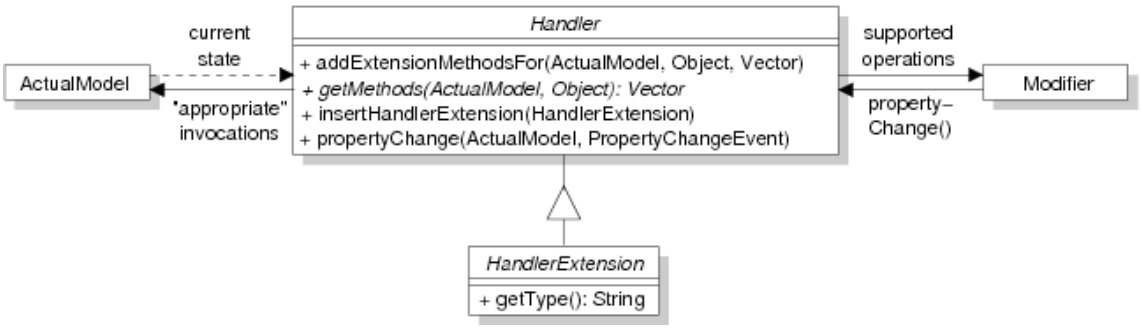

Figure 1: *Request Handler* architecture

The *Modifier* in Figure 1 is the animation effect. It needs to be aware of the operations that can be performed on the selected graphical object. For example, a generic "color changer" animation effect should not offer a "fill color" change operation for unfilled objects, such as texts, points or lines.

Finally, the *RequestHandler* acts as an intermediate object between the two entities. It can be further refined by subclasses of *HandlerExtension*. The *RequestHandler* offers only four methods:

addExtensionMethodsFor looks for existing extension methods that were implemented in one of the (possibly multiple) HandlerExtension objects and adds them to the Vector returned by *getMethods*.

getMethods returns the Vector of all possible concrete transformations for the combination of *ActualModel* and parameter passed in.

insertHandlerExtension is invoked to add a new *HandlerExtension*.

propertyChange is invoked by the *Modifier* whenever a new animation stage has been reached.

Note that of the four methods in the *RequestHandler* interface, only two are actually concerned with Request Handling, while the other two offer extension support.

Negotiating the change of a given property now works as follows. For the sake of clarity, we assume that the user wants to change the *fill color* of a *circle* object, and that both the *ActualModel* circle and the *Modifier* color changer already exist:

1. The *Modifier* invokes *getMethods(myCircle, x)*, where x is a parameter describing the property to be changed. In our example, this can be any arbitrary *java.awt.Color* instance.

2. The *RequestHandler* detects the underlying transformation type, here a color changer, due to the *Color* parameter passed in.

3. The *RequestHandler* queries the *ActualModel* for its current state. In this way, the handler request can detect whether the circle is filled and can therefore change both its outline and fill color, or whether only the outline color can be adapted.

4. The request handler returns a *Vector* of appropriate operation names to the Modifier, allowing the user to choose one. In our example, the Vector may contain the operation names "color", "fill color", and "color + fill color".

5. At some later time, the actual effect is invoked. The *Modifier* determines the current state based on the initial state (the original fill color), the target state (the target fill color) and the current time (what percentage of the effect has been passed), and interpolates the result. This changed value is passed to the request handler using the *propertyChange* method together with the *ActualModel* instance. In our example, the value is converted into an interpolated color along the RBG line between the original and desired target color, according to the percentage of the color change effect that has currently been reached.

6. The request handler extracts the target state and the transformation information from the *PropertyChangeEvent* passed in, which encodes the name of the "property" to be changed as well as its old and new value. It then maps this change into a set of (often nearly trivial) operations on the *ActualModel*. For example, if the method name is "color", it will call the graphical object's *setColor(c)* method, and for "fill color", it will call *setFillColor(c)*.

The *Request Handler* is conceptually similar to the *Adapter* (Gamma et al., 1995, p. 139) and *Mediator* (Gamma et al., 1995, p. 273) design patterns, but differs in a set of key points.

4.4 Consequences

The *Request Handler* pattern has the following consequences:

- The ActualModel classes are decoupled from the Modifier classes. In our example, this means that the graphical objects do not need to be aware of animation as a dynamic change of their internal state. They simply respond to requests to change their internal state (for example, by changing the fill color), and repaint themselves when prompted. Additionally, the animation effects are unaware of the actual objects they modify, and do not need code for handling specific object types.

- The central methods *getMethods* and *propertyChange* are usually trivial to implement. The first needs to figure out, based on the transformation type, what set of operations are possible for the given concrete ActualModel. This is usually very straightforward to implement. The latter method receives both the transformation name generated in *getMethods*, the ActualModel to work on, and the current and target value. Mapping this to appropriate operations on the ActualModel is again usually very simple. For the "fill color" color changer, this operation requires the following steps: recognize that the operation is of a "color change" type; extract the current and target colors from the *PropertyChangeEvent* parameter; extract the method name ("fill color"); based on the method name, decide to call the *setFillColor* model on the ActualModel passed in with the target color.

- As the ActualModel reference is always passed in where it is needed, the *RequestHandler* can be realized as a *Singleton* (Gamma et al., 1995, p. 127) for each graphical object class. This is done in the Animal AV system.

- Using the *HandlerExtension* mechanism, developers can easily add new transformation effect names without needing to modify the code of the original RequestHandler itself.

- The implementation of the *addExtensionMethodsFor* and *insertHandlerExtension* methods can be delegated to a superclass of the actual *RequestHandler* instances, bringing the number of methods to implement down to two per Request Handler.
- If a new graphical object has been implemented, the developer only needs to add a *Request Handler* for this object type. All transformation methods supported by the Handler are then directly usable for the new graphical object - without modifying the code of any transformation object.
- If a new transformation effect has been implemented, the developer has to modify only those Request Handlers that should be able to support the new effect. The effect is then directly usable on the graphical objects without needing to modify their code.
- If a new transformation subtype for a given graphical object shall be supported, the developer only has to add appropriate code to the *getMethods* and *propertyChange* methods. No change of the transformation effect or graphical object classes is necessary. The developer can also put the changes into a new HandlerExtension and register this - in this case, no existing code is touched at all.
- An additional class (the Request Handler) has to be added for each graphical object.

5 Summary and Future Research

In this paper, we have presented two design patterns for AV-related systems. The *Reverse by Fast Forward* can be used to support efficient bidirectional navigation in AV materials. It is also applicable to other materials that match the requirements presented in Section 3 and very easy to implement. The slight delay in execution resulting from the approach is in our experience not or only barely noticeable even when running on older hardware.

The *Request Handler* pattern is used to decouple graphical objects and transformations thereon. It allows the developer to implement new graphical objects without needing to modify existing transformations, or to provide new transformations without modifying the implementation of the graphical objects. After becoming familiar with the underlying concepts, the Request Handler has proven to be highly helpful.

Both patterns have been in active use in the Animal AV system for several years. While grasping them is usually difficult at first for our students implementing new elements for the system, they see the benefits during the implementation phase.

In the future, we hope that other AV researchers will be willing to gather their "best practice" knowledge in the form of design patterns. This shall ultimately help other developers of systems to incorporate tried and proven techniques, and may in the long run even make data exchange between AV systems easier.

References

Ayonike Akingbade, Thomas Finley, Diana Jackson, Pretesh Patel, and Susan H. Rodger. JAWAA: Easy Web-Based Animation from CS 0 to Advanced CS Courses. In *Proceedings of the* 34^{th} *ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2003), Reno, Nevada*, pages 162–166. ACM Press, New York, 2003.

Jay Martin Anderson and Thomas L. Naps. A Context for the Assessment of Algorithm Visualization System as Pedagogical Tools. *First International Program Visualization Workshop, Porvoo, Finland. University of Joensuu Press*, pages 121–130, July 2001.

Bob Boothe. Efficient algorithms for bidirectional debugging. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 299–310, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: http://doi.acm.org/10.1145/349299.349339.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

Martin Odersky and Matthias Zenger. Independently extensible solutions to the expression problem. In *Proc. FOOL 12*, January 2005. `http://homepages.inf.ed.ac.uk/wadler/fool`.

Guido Rößling and Bernd Freisleben. ANIMAL: A System for Supporting Multiple Roles in Algorithm Animation. *Journal of Visual Languages and Computing*, 13(2):341–354, 2002.

Guido Rößling and Thomas L. Naps. A Testbed for Pedagogical Requirements in Algorithm Visualizations. *Proceedings of the 7^{th} Annual ACM SIGCSE / SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE 2002), Århus, Denmark*, pages 96–100, June 2002.

Guido Rößling and Thomas L. Naps. Towards Improved Individual Support in Algorithm Visualization. *Second International Program Visualization Workshop, Århus, Denmark*, pages 125–130, June 2002.

Mads Torgersen. The expression problem revisited. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Oslo, Norway*, pages 123–143, 2004.